Event Handling in the OpenModelica Compiler and Runtime System

Håkan Lundvall, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{haklu,petfr}@ida.liu.se

Abstract

The paper gives an introduction to the problem simulating hybrid DAEs with event-handling using the Modelica language. An implementation in the OpenModelica compiler is presented, and some preliminary results are reported.

1 Introduction

The OpenModelica[1,4,5] compiler is an open source Modelica[2,3] compiler developed at PELAB, Linköping University. So far it has not been possible to simulate hybrid models using the compiler, only pure continuous systems. In order to provide a more complete simulation environment and to support the research carried out at PELAB we have started to add hybrid simulation capabilities to the OpenModelica compiler. One of the research topics that is in need of these capabilities is model checking of hybrid systems in Modelica[6].

2 Conversion of a Modelica Model to a Hybrid DAE

A Modelica model is typically translated to a basic mathematical representation in terms of a flat system of *differential and algebraic equations* (DAEs) before being able to simulate the model. This translation process elaborates on the internal model representation by performing analysis and type checking, inheritance and expansion of base classes, modifications and redeclarations, conversion of connect equations to basic equations, etc. The result of this analysis and translation process is a flat set of equations, including conditional equations, as well as constants, variables, and function definitions. By the term *flat* is meant that the object-oriented structure has been broken down to a flat representation where no trace of the object hierarchy remains apart from dot notation within names

3 Simulation of Hybrid Models Based on Solving Hybrid DAEs

A Modelica *simulation problem* in the general case is a Modelica *model* that can be reduced to a hybrid DAE in the form of equations (1), (2), and (3), together with additional constraints on variables and their derivatives called *initial conditions*.

The initial conditions prescribe initial start values of variables and/or their derivatives at simulation time=0 (e.g. expressed by the Modelica `start` attribute value of variables, with the attribute `fixed` = `true`), or default estimates of start values (the `start` attribute value with `fixed` = `false`).

The simulation problem is *well defined* provided that the following conditions hold:

- The total model system of equations is consistent and neither underdetermined nor overdetermined.
- The initial conditions are consistent and determine initial values for all variables.
- The model is specific enough to define a unique solution from the start simulation time t_0 to some end simulation time t_1.

The initial conditions of the simulation problem are often specified interactively by the user in the simulation tool, e.g. through menus and forms, or alternatively as default `start` attribute values in the simulation code. More complex initial conditions can be specified through `initial equation` sections in Modelica.

4 The Form of the Hybrid DAE System

The hybrid DAE needs to represent both continuous-time behavior and discrete-time behavior. We start by formulating the continous-time part, followed by the discrete-time part.

4.1 Continuous-Time Behavior

Now we want to formulate the continuous part of the *hybrid DAE* system of equations including discrete variables. This is done by adding a vector $q(t_e)$ of *discrete-time variables* and the corresponding predecessor variable vector $q_{pre}(t_e)$ denoted by `pre(q)` in Modelica. For discrete variables we use t_e instead of t to indicate that such variables may only change value at event time points denoted t_e, i.e., the variables $q(t_e)$ and $q_{pre}(t_e)$ behave as constants between events.

We also make the constant vector p of *parameters and constants* explicit in the equations, and make the time t explicit. The vector $c(t_e)$ of condition expressions, e.g. from the conditions of `if` constructs and `when` constructs, evaluated at the most recent event at time t_e is also included since such conditions are referenced in conditional equations. We obtain the following *continuous DAE* system of equations that describe the system behavior *between* events:

$$\begin{aligned} f(x(t),\dot{x}(t),u(t),y(t),t,q(t_e),q_{pre}(t_e),p,c(t_e)) &= 0 \qquad (a) \\ g(x(t),u(t),y(t),t,q(t_e),q_{pre}(t_e),p,c(t_e)) &= 0 \qquad (b) \end{aligned} \tag{1}$$

4.2 Discrete-Time Behavior

Discrete time behavior is closely related to the notion of an event. Events can occur asynchronously, and affect the system one at time, causing a sequence of state transitions. An event occurs when any of conditions $c(t_e)$ (defined below) of conditional equations changes value from `false` to `true`. We say that an event becomes *enabled* at the time t_e, if and only if, for any sufficiently small value of ε, $c(t_e-\varepsilon)$ is `false` and $c(t_e+\varepsilon)$ is `true`. An enabled event is *fired*, i.e., some behavior associated with the event is executed, often causing a discontinuous state transition. Firing of an event may cause other conditions to switch from `false` to `true`. In fact, events are fired until a stable situation is reached when all the condition expressions are `false`.

Discontinuous changes of continuous dynamic variables $x(t)$ can be caused by so-called `reinit` equations in Modelica, which for the sake of simplicity are excluded from the equation representations discussed in this paper.

However, there are also state changes caused by equations defining the values of the *discrete* variables $q(t_e)$, which may change value *only* at events, with event times denoted t_e. Such discrete variables obtain their value at events, e.g. by solving equations in when-equations or evaluating assignments in when-statements. The instantaneous equations defining discrete variables in when-equations are restricted to particularly simple syntactic forms, e.g. *var* `=` *expr*`;`. These restrictions are imposed by the Modelica language in order to easily determine which discrete variables are defined by solving the equations in a when-equation.

Such equations can be directly converted to equations in assignment form, i.e., assignment statements, with fixed causality from the right-hand side to the left-hand side. Regarding algorithmic when-statements that define discrete variables, such definitions are always done through assignments. Therefore we can in both cases express the equations defining discrete variables as *assignments* in the vector equation (1a), where the vector-valued *function* f_q specifies the right-hand side expressions of those *assignments to discrete variables*.

$$q(t_e) := f_q(x(t_e),\dot{x}(t_e),u(t_e),y(t_e),t_e,q_{pre}(t_e),p,c(t_e)) \tag{2}$$

The last argument $c(t_e)$ is made explicit for convenience. It is strictly speaking not necessary since the expressions in $c(t_e)$ could have been incorporated directly into f_q. The vector $c(t_e)$ contains all `Boolean` condition expressions evaluated at the most recent *event* at time t_e. It is defined by the following vector assignment equation with the right-hand side given by the vector-valued function f_e. This function has as arguments the subset of the discrete variables having `Boolean` type, i.e., $q^B(t_e)$ and $q^B_{pre}(t_e)$, the subset of `Boolean` parameters or constants, p^B, and a vector *rel*($v(t)$) evaluated at time t_e, containing the elementary relational expressions from the model. The vector of condition expressions $c(t_e)$ is defined by the following equation in assignment form:

$$c(t_e) := f_e(q^B(t_e), q^B_{pre}(t_e), p^B, rel(v(t_e))) \tag{3}$$

Here $rel(v(t)) = rel(\texttt{cat}(1, x(t), \dot{x}(t), u(t), y(t), t, q(t_e), q_{pre}(t_e), p))$, a `Boolean`-typed vector-valued function containing the relevant elementary *relational expressions* from the model, excluding relations enclosed by `noEvent`(). The argument $v(t) = \{v_1, v_2, ...\}$ is a vector containing all scalar elements of the argument vectors. This can be expressed using the Modelica concatenation function `cat` applied to the vectors, e.g. $v(t) = cat(1, x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p)$. For example, if *rel*($v(t)$) = {$v_1 > v_2$, $v_3 >= 0$, $v_4<5$, $v_6<=v_7$, $v_{12}=133$} where $v(t)$ = {v_1, v_2, v_3, v_4, v_6, v_7, v_{12}}, then it might be the case that $c(t)$ = {$v_1 > v_2$ `and` $v_3 >= 0$, v_{10}, not v_{11}, $v_4<5$ `or` $v_6<=v_7$, $v_{12}=133$}, where v_{10}, v_{11} are `Boolean` variables and v_1, v_2, v_3, v_4, v_6, v_7 might be `Real` variables, whereas v_{12} might be an `Integer` variable.

4.3 The Complete Hybrid DAE

The total equation system consisting of the combination of (1), (2), and (3) is the desired *hybrid DAE* equation representation for Modelica models, consisting of *differential*, *algebraic*, and *discrete* equations.

This framework describes a system where the state evolves in two ways: continuously in time by changing the values of $x(t)$, and by instantaneous changes in the total state represented by the variables $x(t)$, $y(t)$, and $q(t)$. Instantaneous state changes occur at events triggered when some of the conditions $c(t_e)$ change value from `false` to `true`. The set of *state varibles* from which other variables are computed is selected from the set of dynamic variables $x(t)$, algebraic varibles $y(t)$, and discrete-time variables $q(t)$.

Below we summarize the notation used in the above equations, with time dependencies stated explicitly for all time-dependent variables by the arguments t or t_e:

- $p = \{p_1, p_2, ...\}$, a vector containing the Modelica variables declared as `parameter` or `constant` i.e., variables without any time dependency.
- t, the Modelica variable `time`, the independent variable of type `Real` implicitly occurring in all Modelica models.
- $x(t)$, the vector of dynamic variables of the model, i.e., variables of type `Real` that also appear differentiated, meaning that `der()` is applied to them somewhere in the model.
- $\dot{x}(t)$, the differentiated vector of dynamic variables of the model.
- $u(t)$, a vector of input variables, i.e., not dependent on other variables, of type `Real`. These also belong to the set of algebraic variables since they do not appear differentiated.
- $y(t)$, a vector of Modelica variables of type `Real` which do not fall into any other category. Output variables are included among these, which together with $u(t)$ are algebraic variables since they do not appear differentiated.
- $q(t_e)$, a vector of discrete-time Modelica variables of type `discrete Real`, `Boolean`, `Integer` or `String`. These variables change their value only at event instants, i.e., at points t_e in time.
- $q_{pre}(t_e)$, the values of q immediately before the current event occurred, i.e., at time t_e.
- $c(t_e)$, a vector containing all `Boolean` condition expressions evaluated at the most recent *event* at time t_e. This includes conditions from all if-equations/statements and if-expressions from the original model as well as those generated during the conversion of when-equations and when-statements.
- $rel(v(t)) = rel(cat(1, x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p))$, a `Boolean` vector valued function containing the relevant elementary relational expressions from the model, excluding relations enclosed by `noEvent`(). The argument $v(t)$ = {$v_1, v_2, ...$} is a vector containing all elements in the vectors $x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p$. This can be expressed using the Modelica concatenation function `cat` applied to these vectors; *rel*($v(t)$) = {$v_1 > v_2$, $v_3 >= 0$, $v_4<5$, $v_6<=v_7$, $v_{12}=133$} is one possible example.
- $f(...)$, the function that defines the differential equations $f(...) = 0$ in (1a) of the system of equations.
- $g(...)$, the function that defines the algebraic equations $g(...) = 0$ in (1b) of the system of equations.

- $f_q(...)$, the function that defines the difference equations for the discrete variables $q := f_q(...)$, i.e., (2) in the system of equations.
- $f_e(...)$, the function that defines the event conditions $c := f_e(...)$, i.e., (3) in the system of equations.

For simplicity, the special cases involving the `noEvent()`operator and the `reinit()` operator are not contained in the above equations and are not discussed below.

5 Hybrid DAE Solution Algorithm

The general structure of the *hybrid DAE solution algorithm* is presented in Figure 1, emphasizing the main structure rather than details. First, a consistent set of *initial values* needs to be found based on the given constraints, which often requires the solution of an equation system consisting of the initial constraints. Then the hybrid DAE solver checks whether any *event conditions* in when-equations, when-statements, if-expressions, etc. have become `true` and therefore should trigger events. If there is no event, the *continuous DAE solver* is used to numerically solve the DAE until an event occurs or we have reached the end of the prescribed simulation time.

If the conditions for an event are fulfilled, the *event is fired*, that is, the conditional equations associated with the event are activated and solved together with all other active equations. This means that the variables affected by the event are determined, and new values are computed for these variables. Then a new initial value problem has to be solved to find a consistent set of initial values for *restarting* the continuous DAE solver, since there might have been discontinuous changes to both discrete-time and continuous-time variables at the event. This is called the *restart problem*. Of course, firing an event and solving the restart problem may change the values of variables, which in turn causes other event conditions to become `true` and fire the associated events. This iterative process of firing events and solving restart problems is called *event iteration*, which must terminate before restarting the continuous DAE solver.

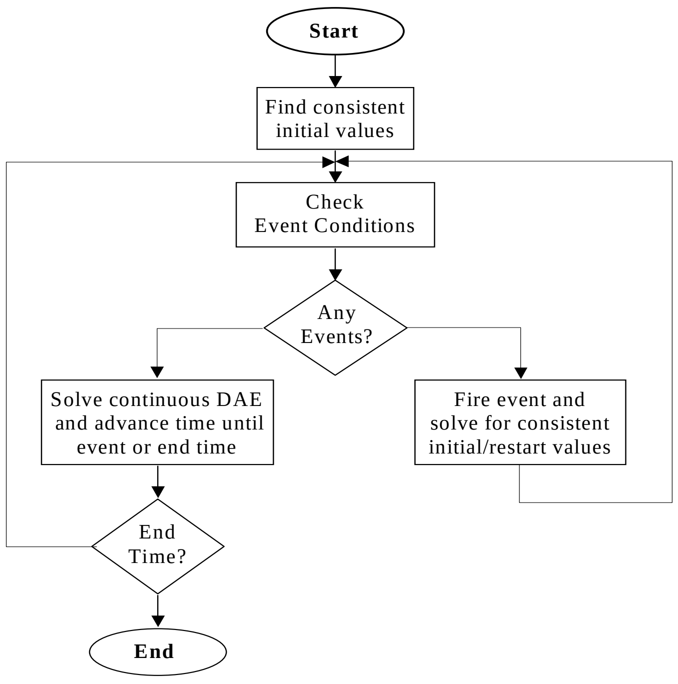
Start
Find consistent initial values
Check Event Conditions
Any Events?
Solve continuous DAE and advance time until event or end time
Fire event and solve for consistent initial/restart values
End Time?
End

Figure 1. General structure of hybrid DAE solution algorithm.

The overall structure of the hybrid DAE solution algorithm is displayed in Figure 1 and summarized below:

1. Solve an *initialization value problem* of finding a consistent set of initial values before starting solution of the continuous part, equation (1), of the hybrid DAE.
2. Solve the *continuous DAE part* (1) of the hybrid DAE using a numerical DAE solver. During this phase the values of the discrete variables q as well as the values of the conditions c from the when-equations, -statements, if-expressions, etc. of the model are kept constant. Therefore the functions

f(...) and *g*(...) in (1) are continuous functions of continuous variables, which fulfills the basic requirements of continuous DAE solvers.

3. During solution of the continuous DAE, all relations *rel*(...) occurring in the conditions *c* are constantly *monitored*. If one of the relations changes value causing a condition to change value from `false` to `true`, the exact time instant of the change is determined, the continuous DAE solution process is halted, and an event is triggered.
4. At an event instant, when an event has been fired, the total system of *active* equations is a mixed set of *algebraic* equations, which is solved for unknowns of type `Real`, `Boolean`, and `Integer`.
5. After the processing of an event, the algorithm continues with step (1) of solving the restart problem of finding a consistent set of initial values. After this step solving the continuous part of the hybrid DAE is restarted if the check in (3) does not indicate new events to be processed in (4).

6 Varying Structure of the Active Part of the Hybrid DAE

Even though the total hybrid DAE system of equations is structurally time invariant, i.e., the set of variables and the set of equations is fixed over time, it is the case that conditional equations in hybrid DAEs can be activated and deactivated. This means that some variables in the state vectors *x* and *q* as well as certain equations can be disabled or deactivated at run-time during simulation, as well as enabled or activated. Such activation or deactivation is caused by events. A disabled variable is kept constant whereas a disabled equation is removed from the total system of active equations that is currently solved. Thus the *active part* of the hybrid DAE can be *structurally dynamic,* i.e., at run-time change the number of *active* variables and equations in the DAE.

7 Finding Consistent Initial Values at Start or Restart

As we have stated briefly above, at the start of the simulation, or at restart after handling an event, it is required to find a consistent set of initial values or restart values of the variables of the hybrid DAE equation system before starting continuous DAE solution process.

At the *start* of the simulation these conditions are given by the initial conditions of the problems (including `start` attribute equations, equations in `initial equation` sections, etc., together with the system of equations defined by (1), (2), and (3). The user specifies the initial time of the simulation, t_0, and initial values or guesses of initial values of some of the continuous variables, derivatives, and discrete-time variables so that the algebraic part of the equation system can be solved at the initial time $t=t_0$ for all the remaining unknown initial values.

At *restart* after an event, the conditions are given by the *new values* of variables that have changed at the event, together with the current values of the remaining variables, and the system of equations (1), (2), and (3). The goal is the same as in the initial case, to solve for the new values of the remaining variables.

In both of the above cases, i.e., at events, including the initial event representing the start of simulation, the process of finding a consistent set of initial values at start or restart is performed by the following iterative procedure, called *event iteration*:

Known variables*:* x, u, t, p

Unknown variables: $\dot{x}, y, q, q_{pre}, c$

```
loop
   Solve the equation system (1) for the unknowns, with $q_{pre}$ fixed;
   if $q = q_{pre}$ then exit loop;
   $q_{pre} := q$;
end loop
```

In the above pseudocode we use the notation q_{pre} corresponding to `pre`(*q*) in Modelica.

8 Detecting Events during Continuous-time Simulation

Event conditions *c* are `Boolean` expressions depending on discrete-time or continuous-time model variables. As soon as an event condition changes from `false` to `true`, the event occurs. It is useful to divide the set of event conditions into two groups: conditions which depend only on discrete-time variables and therefore may change only when an event is fired, and conditions which also depend on continuous-time variables and may change at any time during the solution of the continuous part of the DAE. We call the first group *discrete-time conditions*, and the second group *continuous-time conditions*.

The first group causes no particular problems. The discrete-time conditions are checked after each event when the discrete-time variables might have changed. If some of the conditions change from `false` to `true`, the corresponding events are simply fired.

The continuous-time conditions, however, are more complicated to handle. Each `Boolean` event condition needs to be converted into a continuous function that can be evaluated and monitored along with the continuous-time DAE solution process. Most numerical software, including DAE solution algorithms, is designed to efficiently detect when the values of specified expressions cross zero.

9 Crossing Functions

To be able to detect when `Boolean` conditions become `true`, we convert each continuous-time `Boolean` event condition into a so-called *crossing function*. Such a function of time crosses zero when the `Boolean` condition changes from `false` to `true`. For example, the simple condition expression `y>53` changes from `false` to `true` when `y-53` *crosses* zero from being less than zero to being greater than zero, as depicted in Figure 2. The body of the corresponding crossing function is simply `y-53`.

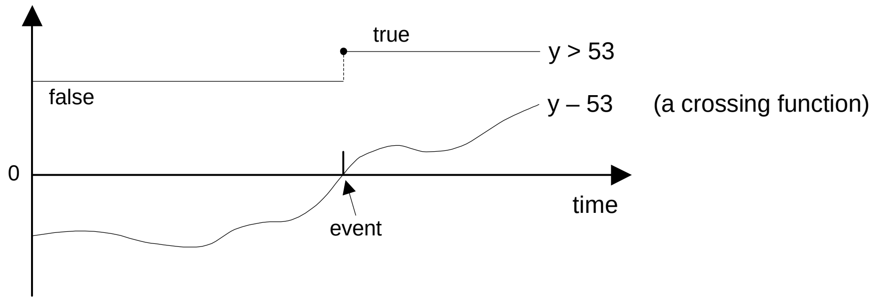

Figure 2. A boolean condition expression `y>53`, with its corresponding crossing function `y-53` that crosses zero at the event, thereby determining the time of the event.

The decision to react on changes of event conditions in Modelica from `false` to `true` rather than from `true` to `false` is arbitrary; it could also have been the other way around.

10 Integrating the Numerical Solver with Event-Handling

In the OpenModelica system a version of DASSL with root finding is used (DASRT)[7]. In order to integrate event handling in the compiler and run-time system, the front-end must produce crossing functions and handlers for the events; the actual search for zero crossings is left to the solver.

The following functions must be made available to the solver:

- `functionDAE_res()` Equations for state variables on residual form.
- `function_ZeroCrossing()` Contains the crossing functions indexed from *0* to *ng-1*, where *ng* is the number of crossing functions.

The sign of the crossing function is chosen in such a way that the function always goes from negative to positive when an event is to be triggered. This is no requirement of the solver, but it is useful when the run-time system checks whether any new events got triggered as a result of variable changes due to the handled event. The runtime system only has to go through the crossing functions and see if any of them has become positive. Before each restart of the solver, the signs of the crossing functions are recalculated so that all the functions are negative. If a crossing function is equal to zero when the solver is to be started, it is disabled by setting it to -1, because the solver cannot handle crossing functions that are zero at startup.

Pseudo code for the simulation loop is shown below.

Integration time variable: *t*
Time of next simulation output: t_{out}
Queue of events: *eventQueue*

```
call DASRT to integrate from $t$ to $t_{out}$;
loop
  if $t$ >= $t_{out}$ then exit loop;
  if DASRT stopped at a root then
    loop
      emit variable values to the result file;
      for each root
        call handleZeroCrossing;
      end for
      emit variable values to the result file;
      call newEventCheck;
      call startEventIteration;
      call DASRT to restart the integration and integrate to $t_{out}$;
      if DASRT did not find roots then exit loop;
    end loop;
  end if;
  emit variable values to the result file;
  $t_{out}$:= $t_{out}$ + $step$;
  call DASRT to integrate from $t$ to $t_{out}$;
end loop;
emit variable values to the result file;
end;

function startEventIteration
  loop
    if $eventQueue$ is empty then exit loop;
    $event$ := pop $eventQueue$;
    if $event$ is a boolean variable change then
      call handleEvent;
    else
      call handleZeroCrossing;
    end if;

    call newEventCheck;
  end loop;
end function;
```

11 Code Generation

In the equations sorting algorithm we assume that the equations conform to certain rules which simplifies the sorting:

1) There are no if-equations with non-constant conditions. Such equations are first transformed to equations of the form `0 = if cond then <truepart> else <falsepart>;`. This way they can be treated as regular equations.

2) The conditional expressions of when-clauses only contain boolean variables. If a condition involves a relation expression a help variable is introduced along with an equation binding it to the expression it replaces. This way when the solver stops as a result of a crossing function becom-

ing positive. Handling routines only has to set the appropriate discrete variable and then the event iteration mechanism handles the triggering of the when-equations.

When dealing with hybrid simulations one could make a distinction between time-events and state-events. By time-events we mean events triggered by expressions not depending, directly or indirectly, on any state variables. The triggering time of such events can be calculated beforehand and the root finder of the solver need not be used. This is more efficient, but in this first version we do not make this distinction.

Apart from the functions listed in section 10 the simulation code generated by OpenModelica contains the following functions:

• `functionDAE_output()`	Equations for output variables
• `handleZeroCrossing()`	Called by the simulation loop when the solver has stopped as a result of a crossing function passing zero.
• `handleEvent()`	Called by the event iteration loop Whenever a boolean variable has become true as a result of a zero crossing or as a result of another event.
• `newEventCheck()`	Called once for each iteration in the event iteration loop to check if any new event was triggered.

In the first step of the equation sorting part of the compilation, all equations and variables are gathered. In this stage all equations appearing inside when clauses are checked to see if they conform to the requirements of when-clauses and all variables that appear as left hand sides of these equations are marked as discrete. These equations are considered during the rest of the equation sorting, but instead of being output in the functions used by the DAE solver (`functionDAE_res()`and `functionDAE_output()`), they are put in the `handleEvent()`-function and are therefore only calculated at events instead of for each iteration of the solver.

Next, every expression in the model, apart from those appearing inside `noEvent()`, etc, are searched for relation expressions, e.g., $x > 5$. Each such expression, labeled with a list of all equations in which it occurs, is added to a list of zero crossings (*rel*). If *rel* contains more than one element with identical relation expressions, those elements are merged together by appending the lists of equations in them. Each element in *rel* generates on crossing function in `zeroCrossing()` and a section in `handleZeroCrossing()` in which the equations assigned to the relation is output. All variables depending on variables updated in `handleZero-Crossing()` are also updated.

In the `newEventCheck()`-function a test of the form “**`if`** `(y != pre(y))`” is generated for each discrete variable *y*. If the variable *y* is in the condition expression of a when-clause then the index of that when-clause is placed on a queue holding the events yet to be handled. If there are variables depending on *y* according to the sorting of the equations, then these variables are updated. If this causes a zero crossing to change its sign then the index of that zero crossing is placed in the event queue.

The event iteration starts by checking for any new events that has been fired as result of the crossing function passing zero. This is done by calling newEventCheck() in the generated simulation code followed by a call to `function_ZeroCrossing()`. For each crossing function reporting a positive result the index of that crossing function is placed on the event queue. Then the first event in the queue is handled and the check for new events is carried out again. This continues until the event queue is empty. At this time new consistent start condition is calculated and the solver is restarted.

Since each unique relation expression generate its own zero crossing function it is possible to write a model that causes the integrator to stop even though the no event actually occur, *e.g.*, the code; “**`when`** `b` **`or`** `x > 3` **`then`** …”, would cause an unnecessary interruption of the integrator if the variable `b` is already `true`. It would be possible detect such situations and disable the zero crossing function for when it would not influence the result. We have chosen not to detect this for now to simplify the implementation.

12 Measurements and Evaluation

12.1 Bouncing Ball

As a small test case for the implementation we have used a model of a bouncing ball.

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
equation
  impact = h <= 0;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {impact, h <= 0 and v <= 0} then
    v_new = if edge(impact) then -e*pre(v) else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;
```

Due to numerical effects it is necessary to separately handle the case where the heights of the bounces become smaller than the tolerance of the simulation. Otherwise, the ball could “fall through the floor”.

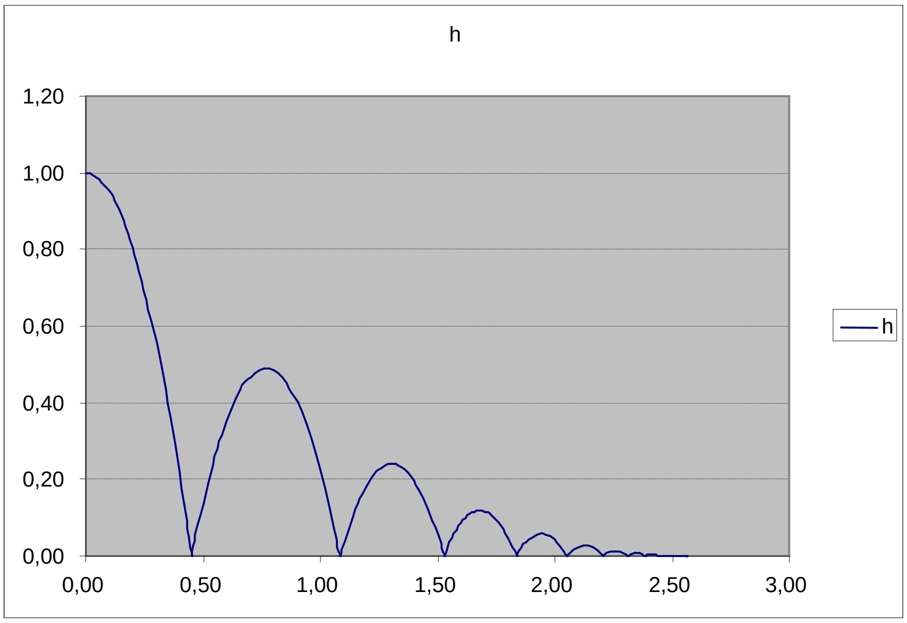

Figure 3. Plot of the height of the ball as a function of time.

When we compare the times (t_e) of the bouncing events to the analytically calculated times denoted t_e they differ in time from the analytically calculated by more than $5.2 \cdot 10^{-8}$ s.

12.2 Event Iteration

The following model tests the event iteration. When *x* reaches two a chain of events starts that end with the setting of *z* to true. The iteration involves both when conditions being set directly as in the case of *y*, and indirectly by setting *a* to 2.0 causing *h2* to becoming true. The whole chain of events takes place in the same iteration without the continuous solver ever having to restart in the middle.

```
model EventIteration
  Real x(start=1.0), dx;
  discrete Real a(start=1.0);
  Boolean y(start=false), z(start=false);
  Boolean h1, h2;
equation
  der(x) = dx;
  dx = a*x;
  h1 = x >= 2;
  h2 = dx >= 4;

  when h1 then
    y = true;
  end when;
  when y then
    a = 2.0;
  end when;
  when h2 then
    z = true;
  end when;
end EventIteration;
```

13 Conclusions and future work

In this paper we have presented an overview of our implementation of discrete event handling in the OpenModelica compiler. In the test models we show that event iteration work as expected and that the simulation results correspond well to analytically calculated results.

The implementation is however not yet complete. In the future we wish implement separate handling of events where the time of the event can be calculated before hand, so that the integrator does not have to search for the roots. When time events are handled we plan to use the OpenModelica compiler as a basis for research in the field of model checking of hybrid systems.

14 Acknowledgements

This work has been supported by Swedish Foundation for Strategic Research (SSF) in the ECSEL graduate school.

References

[1] Peter Fritzson, *et al*. The Open Source Modelica Project. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany See also: http://www.ida.liu.se/projects/OpenModelica.

[2] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004.

[3] The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. http://www.modelica.org.

[4] The OpenModelica Users Guide, version 0.2, April 2005. www.ida.liu.se/projects/OpenModelica

[5] The OpenModelica System Documentation, version 0.2, April 2005. www.ida.liu.se/projects/OpenModelica

[6] Håkan Lundvall, Peter Bunus, Peter Fritzson. Towards Automatic Generation of Model Checkable Code from Modelica. In *Proceedings of the 45th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2004)*, 23-24 September 2004, Copenhagen, Denmark.

[7] K. E. Brenan, S. L. Campbell, and L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Elsevier, New York, 1989.